1 实验目的与方法

1.1 实验目的

1.1.1 词法分析器

(1) 加深对词法分析程序的功能及实现方法的理解；

(2) 对类 C 语言的文法描述有更深的认识，理解有穷自动机、编码表和符号表在编译的整个过程中的应用；

(3) 设计并编程实现一个词法分析程序，对类 C 语言源程序段进行词法分析，加深对高级语言的认识。

1.1.2 语法分析

(1) 深入了解语法分析程序实现原理及方法；

(2) 理解 LR(1)分析法是严格的从左向右扫描和自底向上的语法分析方法。

1.1.3 典型语句的语义分析及中间代码生成

(1) 加深对自顶向下语法制导翻译技术的理解与掌握；

(2) 加深对自底向上语法制导翻译技术的理解与掌握；

(3) 巩固对语义分析的基本功能和原理的认识，理解中间代码生成的作用。

1.1.4 目标代码生成

(1) 加深对编译器总体结构的理解与掌握；

(2) 加深对汇编指令的理解与掌握；

(3) 对指令选择，寄存器分配和计算顺序选择有较深的理解。

1.2 实验方法

操作系统：Windows10

IDE：Visual Studio 2022

程序语言：C#

目标框架：.NET 6.0

程序类型：控制台应用程序

2 实验内容及要求

2.1 词法分析器

编写一个词法分析程序，读取代码文件，对文件内自定义的类 C 语言程序段进行词法分析。处理 C 语言源程序，过滤掉无用符号，分解出正确的单词，以二元组形式存输出放在文件中。

(1) 词法分析程序输入：以文件形式存放自定义的类 C 语言程序段;

(2) 词法分析程序输出：以文件形式存放的 Token 串和简单符号表；

(3) 词法分析程序输入单词类型要求：输入的 C 语言程序段包含常见的关键字，标识符，常数，运算符和分界符等。

2.2 语法分析

(1) 利用 LR(1)分析法，设计一个语法分析程序，对输入单词符号串进行语法分析；

(2) 输出推导过程中所用产生式序列并保存在输出文件中。

2.3 典型语句的语义分析及中间代码生成

(1) 针对自顶向下或者自底向上分析法中所使用的文法，在完成语法分析的基础上为语法正确的单词串设计翻译方案；

(2) 利用该翻译方案，对所给程序段进行分析，输出生成的中间代码序列和符号表，并保存在相应文件中；

(3) 中间代码可选三地址码的三元式表示或者四元式表示。

2.4 目标代码生成

(1) 将中间代码所给的地址码生成目标代码；

(2) 写出代码序列表；

(3) 减少程序与指令的开销，进行部分优化。

3 实验总体流程与函数功能描述

3.1 实验流程

实验总体流程：词法分析器——自底向上的 LR(1)语法分析器——基于语法分析实现自底向上的语法制导翻译（语义分析）和中间代码生成——目标代码生成

词法分析器实现流程：确定编码表和有穷自动机——根据编码表和有穷自动机完成词法分析程序——输入源代码，输出符号表和单词串

语法分析器实现流程：确定文法——使用软件“编译工作台”，根据文法生成 LR 状态分析表——根据文法编码预处理 LR 状态分析表——完成自底向上的语法分析程序——输入 LR 状态分析表和单词串，输出产生式序列

语义分析和中间代码生成实现流程：修改语法分析程序为自底向上的语法制导翻译程序——输入 LR 状态分析表和单词串，填写符号表，输出产生式序列、符号表（新）和三地址码（三元式）

目标代码生成实现流程：根据 x86 汇编和对寄存器使用、访存次数的考量完成目标代码生成程序——输入符号表和三地址码，输出目标代码

3.2 关键函数功能描述

3.2.1 词法分析器

源代码见 LexicalAnalyzer.cs

Analyze()：实现有穷自动机，输入源代码，输出符号表和单词串

StrAnalyze()：字符和字符串单词的提取

TokenMatch()：标识符、整数、浮点数单词的提取

3.2.2 语法分析、语义分析和中间代码生成

源代码见 GrammarAnalyzer.cs

Preprocess()：根据文法编码预处理 LR 状态分析表文件

InitTable()：根据 LR 状态分析表文件初始化 action 和 goto 表

SearchAction()：在 action 表中查找当前状态和终结符对应的动作（移进/归约）和转移状态

SearchGo2()：在 goto 表中查找当前状态和非终结符对应的转移状态

InputPre()：以单词串中分号将输入的单词串分为一条条等待语法分析的语句（单词串队列）

SemAnalyze()：自底向上的语义分析，填写符号表，输出三地址码，函数返回语义信息

Analyze()：自底向上的语法制导翻译。输入一条语句的单词串，利用 action 和 goto 表、状态栈、单词编码栈和语义信息栈（存放语义相关的单词值）进行语法和语义分析，语法分析有移进、归约、接受语句、语句出错四个模块，其中语义分析在归约时进行。输出产生式序列、符号表和三地址码

3.2.3 目标代码生成

源代码见 ObjGen.cs，x86 汇编，寄存器只使用 EAX, ECX, EDX

ImmCal()：常数运算直接用结果替换三地址码中的对应内容

Assign()：赋值，左元为已声明类型的标识符，右元为立即数或寄存器号，如果某寄存器值归属被赋值的变量，则该寄存器值归属置空

RegChoose()：选出一个满足下列条件之一的寄存器淘汰（刷新）存放值（优先返回给定寄存器序列中第一个满足条件的寄存器）：

(1) 空归属寄存器;

(2) 从输入的起始三地址码序号起，不再被使用到其存放值的寄存器，存放值过时的寄存器被使用则视为未被使用；

(3) 从输入的起始三地址码序号起，最远被使用到其存放值的寄存器

当选出的寄存器中存放的值是会被使用到的中间结果时，先把该寄存器的值存放在最后一个标识符后（基址）的某一地址内存中（偏移量由该中间结果的类型和三地址码序号计算而来），再淘汰该寄存器值

更新寄存器的值前都将使用 RegChoose()函数，输入的起始三地址码序号（一般是当前或下一个）和给定寄存器序列视情况而定

OpCalGen()：满足交换律的四则运算目标代码生成

首先，检查左元的值是否在寄存器中，若不在，则为(1) 立即数；(2) 中间结果，（生成的目标代码为）按其地址（计算方法见 RegChoose()中的描述）从内存中取到淘汰寄存器中；(3) 标识符代表的变量，查符号表，按其地址从内存中取到淘汰寄存器中

右元同理

然后，若左元（或右元，如果有一个是立即数，则此处仅为非立即数）归属寄存器值可以淘汰，则生成一个所需的运算指令，结果存于该寄存器中；否则，先将左元移到可淘汰寄存器，然后再生成一个所需的运算指令，结果存于该寄存器中

OpCalGen1()：不满足交换律的四则运算目标代码生成
与 OpCalGen()函数不同的是，当左元是立即数时，需要将立即数移进淘汰寄存器，再做运算；当左元归属不是而右元归属是淘汰寄存器时，与 RegChoose()函数相似，只是忽视其选择条件，直接选左元归属寄存器并进行后续操作（若已生成右元归属淘汰的访存代码，则先撤销该访存代码，再进行后续操作），再做运算

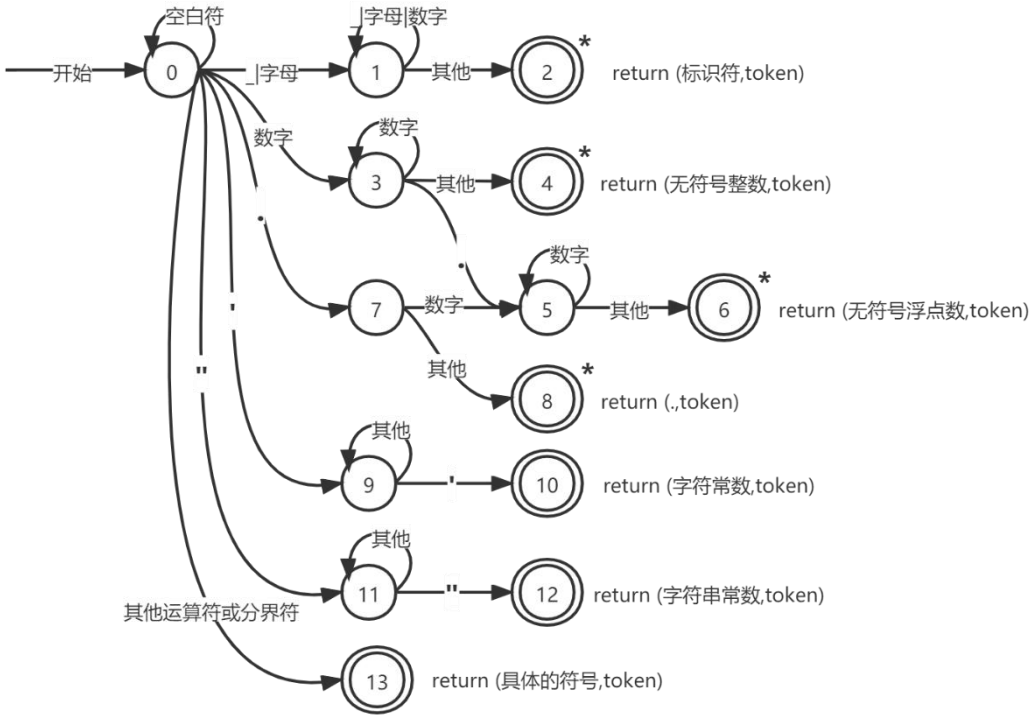
Generate()：目标代码生成。遍历三地址码，访存时查符号表。当左、右元皆为常数时，调用 ImmCal()函数；赋值算符右元为变量或中间变量，如果值不在寄存器中，则先访问对应地址的内存，取值到淘汰寄存器，再进行赋值；当算符为赋值或四则运算算符时，调用上述对应的函数

3.3 回答问题和实验思考

3.3.1 词法分析器

(1) 文法见文法.txt

(2) 有穷自动机如下图所示：



(3) 编码表见编码表.xlsx

(4) 符号表是一个链表，每个结点存放一个三元组(string 标识符,int 类别,string 属性)

(5) 词法分析算法基本根据上图的有穷自动机来实现。更具体的，标识符会判断是不是关键字，不是关键字会查符号表，是新的标识符则加入符号表；字符常数从第一个单引号起，无换行直至第二个单引号且该单引号前无转义符，不满足则退回到第一个单引号并提取为单词，字符串常数类似；行注释和多行注释跳过

(6) 最好是不用空格区分单词，但输入标识符和关键字之间必须要有空格

(7) 词法分析程序作为独立子程序，便于修改词法分析逻辑而不影响其他部分的程序，也方便和其他程序组装在一起实现更加复杂的功能

3.3.2 语法分析

(1) 文法见文法.txt

(2) 由编译工作台导出识别文法活前缀的有穷自动机，见识别文法活前缀的有穷自动机图.docx

(3) 由编译工作台导出所给文法的 LR(1)分析表，见 LR 状态分析表_原表.xlsx

(4) LR(1)分析表分为 action 和 goto 表，分别用结点为四元组(int 当前状态,int 终结符, int 动作,int 转移状态)和结点为三元组(int 当前状态,int 非终结符,int 转移状态)的链表存储

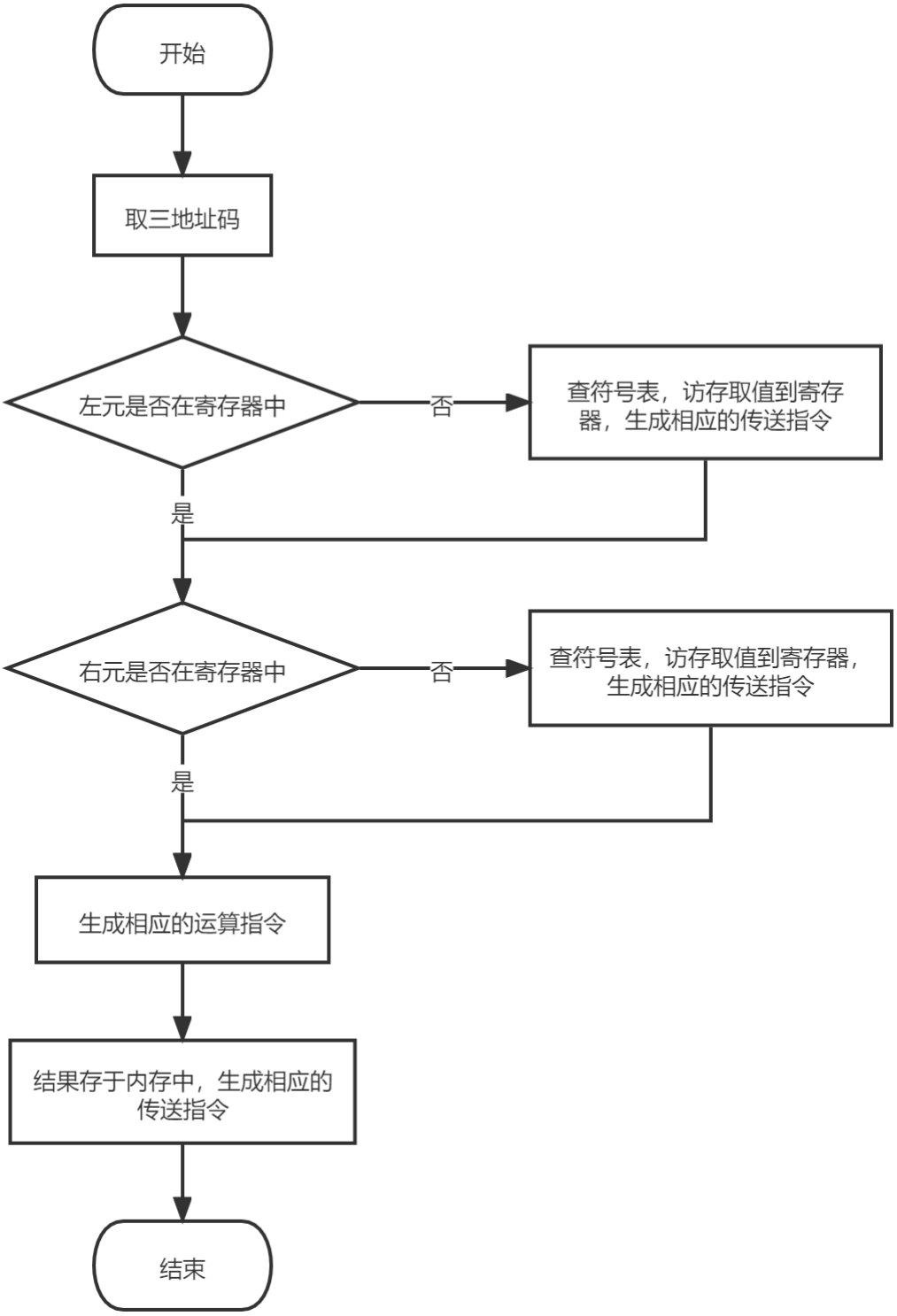
(5) 语法分析的主要算法见本报告 3.2.2 Analyze()函数功能描述

3.3.3 典型语句的语义分析及中间代码生成

(1) 翻译方案见本报告 3.2.2 Analyze()函数功能描述
(2) 在语法分析所用的状态栈和单词编码栈的基础上加一个语义信息栈，用于语义分析
(3) 中间代码以三地址码的形式存储于结点为三元组(string 算符,string 左元,string 右元)的链表中，并填写词法分析输出的符号表

3.3.4 目标代码生成

(1) 程序开销见实验结果与分析
(2) 一个简单的代码生成器流程图如下图所示：



对于该流程，可以通过调整对寄存器的使用方案和调整运算指令的生成方案来省去一些访存指令，但寄存器间的传送指令会略微增加，具体实现见本报告 3.2.3 部分

4 实验结果与分析

程序和输入输出文件的详细说明见 readme.txt

词法分析器：

输入源代码 lex-input-code.txt

```
int a, b, c;
a = 8;
b = 5 * a;
c = 10 / 3;
int result = a * b + ( a - b ) + c + ( a - b ) + a * b;
a = 10;
result = a * b + ( a - b );
```

输出符号表 lex-output-tableWords.txt（标识符，类型，属性）

```
(a,-1,?)
(b,-1,?)
(c,-1,?)
(result,-1,?)
```

输出单词串 lex-output-token.txt（单词码，单词值）（报告仅截取一部分，详细见文件）

```
(22,int)
(1,a)
(73,,)
(1,b)
(73,,)
(1,c)
(80,;)
(1,a)
(60,=)
(2,8)
(80,;)
```

……

正确填写了符号表，区分了标识符和关键字，输出的单词串每个元组皆为目标单词和其在编码表中对应的编码，测试用例输出结果无误

语法分析：

输入来自词法分析和给定的 LR 状态分析表

输出产生式序列 gram-output.txt（报告仅截取一部分，详细见文件）

```
T -> id
S -> int T
T -> id
S -> S , T
T -> id
S -> S , T
E' -> S
Accept a sentence.
C -> num
B -> C
A -> B
E -> id = A
E' -> E
Accept a sentence.
C -> num
B -> C
C -> id
B -> B * C
A -> B
E -> id = A
E' -> E
Accept a sentence.
```

……

产生式及其说明见文法.txt，“Accept a sentence”表示输入的语句通过语法分析，符合给定的语法规则。经检验，测试用例中每条语句对应的产生式序列输出无误

语义分析和中间代码生成：

输入来自词法分析和语法分析（和语法分析结合）

输出符号表 sem-output-tableWords.txt（标识符，类型，地址，符号种类，扩展属性）

```
(a,22,0,变量,NULL)
(b,22,4,变量,NULL)
(c,22,8,变量,NULL)
(result,22,12,变量,NULL)
```

输出三地址码 sem-output-3Addr.txt（算符，左元，右元）

```
(=,a,8)
(*,5,a)
(=,b,(1))
(/,10,3)
(=,c,(3))
(*,a,b)
(-,a,b)
(+,(5),(6))
(+,(7),c)
(+,(8),(6))
(+,(9),(5))
(=,result,(10))
(=,a,10)
(*,a,b)
(-,a,b)
(+,(13),(14))
(=,result,(15))
```

三地址码元组中的(1)表示下标为 1 的三地址码运算结果

符号表填写和输出的三地址码无误。

目标代码生成：

输入来自语义分析和中间代码生成

输出目标代码 obj-output.txt（x86 汇编 仅代码段的汇编代码）

```
mov dword ptr [a],8
mov eax,dword ptr [a]
mov ecx,eax
imul ecx,5
mov dword ptr [b],ecx
mov dword ptr [c],3
```

……（此处仅截取一部分，详细见文件）

源代码中变量应先声明后使用，否则目标代码会缺失访存取值到寄存器的指令，因为未声明的变量不在符号表中

接下来对整个编译器最终输出的目标代码，根据最初输入的源代码，逐语句进行分析

int a, b, c;

声明语句无代码段目标代码

a = 8;

mov dword ptr [a],8

将 8 传送到地址为[a]（取地址）的内存（双字写，因为 int 类型是 32 位）

b = 5 * a;

mov eax,dword ptr [a]

mov ecx,eax

imul ecx,5

mov dword ptr [b],ecx

仅使用 eax、ecx、edx 三个寄存器，imul 为带符号数乘法运算指令。首先寄存器中没有 a 值，则访存取值。根据前述的寄存器淘汰规则，eax 中的 a 值会被用到，而 ecx 为空，所以将 eax 中的 a 值传入 ecx，然后参与运算，并由 ecx 保存运算结果

c = 10 / 3;

mov dword ptr [c],3

常数运算，直接替换为运算结果，再生成目标代码

int result = a * b + (a - b) + c + (a - b) + a * b;

mov edx,dword ptr [b] ;寄存器无 b 值，edx 为空，移进

mov ecx,eax ;ecx 改 5*a 值（等效 b 值，上一行可优化，为简化实现而忽略）为 a 值

imul ecx,edx ;因为 ecx 的 5*a 值不被使用，所以移入 a 参与 a*b 运算并保存运算结果

sub eax,edx ;eax 保存 a-b 值

```
mov edx,ecx    ;edx 满足淘汰条件，将 ecx 的 a*b 值传入 edx

add edx,eax    ;edx 保存 a*b+(a-b)值

mov dword ptr [result+24],ecx      ;ecx 淘汰值 a*b 会被使用，存到临时变量地址

mov ecx,dword ptr [c]

add edx,ecx    ; edx 旧值不使用，更新为 a*b+(a-b)+c 值

;上一行若改为 add edx,dword ptr[c]可以不淘汰 ecx 旧值，减少访存次数

;此处作实现上的简化处理，不作优化

add edx,eax    ; edx 旧值不使用，更新为 a*b+(a-b)+c+(a-b)值

mov eax,dword ptr [result+24]      ;eax 的 a-b 值不再被使用，淘汰

add edx,eax    ;edx=a*b+(a-b)+c+(a-b)+a*b

mov dword ptr [result],edx    ;result=edx，完成赋值
```

如果运用加法交换律和结合律，可以提取更大的公共子表达式，节省指令与访存次数。但此处作实现上的简化处理，不作上述优化

```
a = 10;

mov dword ptr [a],10



result = a * b + ( a - b );

mov eax,dword ptr [a]

mov ecx,dword ptr [b]

mov edx,eax

imul edx,ecx   ;edx=a*b
```

```
sub eax,ecx     ;eax=a-b

add edx,eax     ;edx=a*b+(a-b)

mov dword ptr [result],edx     ;result=edx
```

由源代码生成的目标代码（x86 汇编 代码段代码）无误。共使用 17 次 mov 指令，5 次 add 指令（源代码共 5 次加法），2 次 sub 指令（源代码共 3 次减法），3 次 imul 指令（源代码共 4 次乘法），0 次 idiv 指令（源代码共 1 次除法），进行了 13 次访存

在 visual studio 2022 中选择 x86 平台调试该 c++源代码（用例符合 c++语法规则），反汇编结果对应上述内容的数字序列为 12，5，3，4，0，20（未经编译器优化），即相对于未优化的代码，生成的目标代码多 5 次 mov 指令，少 1 次 sub 指令，少 1 次 imul 指令，少 7 次访存

5 实验中遇到的困难与解决办法

(1) 中间代码生成的三地址码中，临时结果的指向问题。如 a=b+c; d=b+c;在三地址码中会指向同一个 b+c 的三地址码吗

如果 b 或 c 值未更新，可以指向同一个；有更新，必须重新生成一个新的 b+c 三地址码。为了简化实现，最终选择在一条语句中只算一次并指向同一个，其他条语句中用到 b+c 需重新算一次。故该情况的实现结果是不会指向同一个 b+c 的三地址码

(2) 寄存器使用问题

参考了 c++反汇编代码，只使用 eax，ecx，edx 三个寄存器，联系操作系统调页的想法来使用寄存器，设计寄存器使用方案来减少访存次数（但增加了 mov 指令的使用次数）